Domain Expert-Directed Program Optimizations for Accelerated Performance on Heterogeneous Multi-core Processors

(Reference No. AOARD-11-4029)

Final Report (December 2013)

By Dr. Pen-Chung Yew and Dr. Bo-YinYang

1. Project Summary

The proposed work on program optimizations directed by domain experts for accelerated performance on heterogeneous multi-core processors includes two major components. One is focussed on computational fluid dynamic (CFD) applications that includes a front-end program transformation tool to assist domain experts to optimize their CFD codes. The other component is on cryptograph applications that includes a prototype compiler (based on *qhasm* compiler [1]) to generate high-performance and correct code for targeted multi-core platforms.

It is well-known that general-purpose optimizing compilers provided by vendors cannot carry out damain-specific program optimizations and code generation required to achieve a high performance obtainable by domain-specific application programs. Domain experts armed with the knowledge of their own programs can tune their programs with program optimizations and generate optimized code tailored to their programs targeting specific platforms. Some of these program optimizations and code generation are very labor intensive and error-prone. Domain-specific compiler tools that can work with domain experts to optimize and generate code under the direction of domain experts can yield a much higher performance on targeted multicore systems than vendor-provided general-purpose compilers.

In this project, tools have been built to tackle two important application domains, namely CFD and cryptography applications. This final report consists of two parts: one on CFD and the other on cryptographic applications. In Section 2, a pre-compiler, called CFD Builder, is presented. It can transform CFD source programs to an optmized form that will allow vendor compilers to gerenate high-performance binary codes. In Section 3, a domain-specific compiler targeting cryptographic applications is presented. It can generate optimized codes that can be proven correct by using formal methods.

2. CFD Builder – A Domain-Specific Pre-compiler for CFD Applications

CFD Builder is intended as an additional layer in the software stack. It is a tool for fast and effective delivery of coding techniques that can deliver high performance for CFD algorithms that are made to fit the patterns CFD Builder exposes. The intent is to allow library writers to pick and choose from the patterns that the tool provides. CFD Builder is a tool to assist in creating a library of routines for use in CFD applications. It incorporates a set of useful and effective code transformations, but leaves to the code developer the choice of which numerical algorithms to transform. The main motivation to build CFD Builder is to achieve performance greater than 20% of processor peak for CFD applications. It is a tool that can handle the very tedious parts of the task but allows the programmer to have as much control as possible over the numerical algorithm and over easily specified parameters like the ones for the data layout. CFD Builder is a productivity tool that is meant to be transparent about its workings.

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **17 JAN 2014**	2. REPORT TYPE **Final**	3. DATES COVERED **15-08-2011 to 14-08-2012**

4. TITLE AND SUBTITLE **Domain expert-directed program optimizations for accelerated performance on heterogeneous multi-core processors**	5a. CONTRACT NUMBER **FA23861114092**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) **Pen-Chung Yew**	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Academia Sinica,128 Academia Road, Section 2, Nankang,Taipei 115,Taiwan,TW,115**	8. PERFORMING ORGANIZATION REPORT NUMBER **N/A**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) **AOARD, UNIT 45002, APO, AP, 96338-5002**	10. SPONSOR/MONITOR'S ACRONYM(S) **AOARD**
	11. SPONSOR/MONITOR'S REPORT NUMBER(S) **AOARD-114092**

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The proposed work on program optimizations directed by domain experts for accelerated performance on heterogeneous multi-core processors includes two major components. One is focused on computational fluid dynamic (CFD) applications that includes a front-end program transformation tool to assist domain experts to optimize their CFD codes. The other component is on cryptograph applications that includes a prototype compiler (based on qhasm compiler [1]) to generate high-performance and correct code for targeted multi-core platforms.

15. SUBJECT TERMS
computer system architecture, Computers

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**15**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

2.1 Key Techniques Used in CFD Builder

2.1.1 Cache blocking through miniaturization of domain decomposition

In CFD, a simulation over a large domain is usually broken down into smaller sub-domains and distributed across the nodes. The sub-domains can be updated independently. However to do so, they need boundary information from their neighbors at every single time-step update. The sub-domains are augmented to send and receive the boundary information with their neighbors. They are then updated from the boundary information received. This approach has overheads. It performs redundant computations for the boundary regions at every sub-domain. It also needs to pass messages between the sub-domains. This form of parallelization is called *domain decomposition*, and is common knowledge in the CFD community. A key consideration in the domain decomposition approach is that the sub-domains fit in the main memory in every processing node. The disks are only used for storing the results of the computations and the checkpoint information. For performance reasons, they are not involved in the computation.

CFD Builder extends this notion of computing out of the main memory to computing out of the on-chip storage with a data layout called *briquettes*. A typical *briquette* is 4^3 grid cells. Its size is a parameter that can be varied. In effect, the data layout is a miniaturization of the domain decomposition technique for *cache blocking*.

Cache blocking through the miniaturization of domain decomposition makes programming for the briquette data layout easier. In domain decomposition approach, the sub-domains are augmented with the boundary region to simplify programming. All the necessary information for a time-step update is contained within the augmented sub-domain. The computation then can work with the augmented sub-domain and be completely oblivious of the other sub-domains. If an augmented briquette with the boundary information can be constructed then the code to update this augmented briquette is the same as the code to update an augmented sub-domain.

The overheads of domain decomposition are more pronounced at smaller scales. The programming convenience of miniaturizing the domain decomposition approach comes at the cost of redundant copies involved in constructing the augmented briquettes and redundant computations between briquettes as illustrated in Figure 1. Grid update patterns are not about miniaturizing the domain decomposition approach. The intent here is to only motivate the data layout with a data structure, called briquette, and an easy way to program for the data layout. A *pipelining-for-reuse* technique implemented in CFD Builder converts this simpler program expression into another code expression, which eliminates the overhead to update the briquette data layout.

2.1.2 The main data structures used - briquettes

From cache blocking technique above, the entire workspace is set to fit in cache, and enough work must be done in that workspace to amortize the cost of main memory data transmission to cache. To do so, a blocked data structure, called a *briquette,* is used where each block must fit easily into the cache. The size of a briquette is determined by many factors. The most important among them is the width of the SIMD engines on a multicore processor. It imposes a minimum size on the briquette.

All problem states are packaged in a briquette for efficient prefetching so that each briquette is many cache lines long. Efficient memory accesses can be performed with no strides, by reading and writing to the main memory in quanta of the briquettes. The MPI messages are constructed out of briquettes as well. The number of boundary (ghost) cells is another consideration for the size of a briquette to avoid sending too much excess data in form of partially filled briquettes.

It is therefore more efficient to update one entire briquette in a single traversal of some fundamental loop over the numerical algorithm's equations. This is like cache blocking as discussed earlier. However, cache blocking through miniaturizing domain decomposition technique suffers from a lot of redundant computations and redundant copies. Instead, multiple briquettes are updated at the same time to reuse computation and data copies – a technique called *pipelining-for-reuse.*

2.1.3 Pipelining-for-resuse

Pipelining-for-reuse is a technique to avoid redundant computations and redundant copying in updating the data layout. In the input code expression that performs cache blocking through domain decomposition technique, the boundary region of one briquette is the real region for its neighboring briquette. The augmented regions of neighboring briquettes overlap. Therefore, redundant copy operations can be avoided if the grid planes in the overlapped region can be repositioned to correspond to the augmented briquette currently being computed. The planes are repositioned by indexing them indirectly with a set of revolving indices. The planes are stored in a circular buffer. Any element can be indexed in the buffer with the revolving indices. The read operations do not delete any elements. Instead, the write operations are used to overwrite the old values. All data copies can thus be reused when the briquettes are processed in a line.

Redundant computations are avoided by reusing the partial results of computations between the augmented briquettes of the input code. However, the concept of augmented briquettes does not exist in the pipelined code expression. The partial results correspond to the temporary arrays in the computation region. Each grid plane of the problem state is updated in multiple stages. There is a workspace of temporary arrays associated with each grid plane. The size of the workspace is much larger when compared to the plane. We want to minimize the requirement for on-chip data storage. Consistent with good utilization of the SIMD engine, it can be done by constructing code to perform a pipelined update of a single grid plane. On pipelining, it has a circular buffer of the workspaces.

The number of elements in the circular buffer is smaller than the number of grid planes in an augmented briquette. It is equal to the twice number of the boundary cells plus one for the grid plane. In this respect, it is not only a work saving optimization but a memory saving optimization too. An important point is that we store each partial result in its own individual circular buffer of grid planes. Here, a workspace represents all the grid planes of partial results corresponding to its stage.

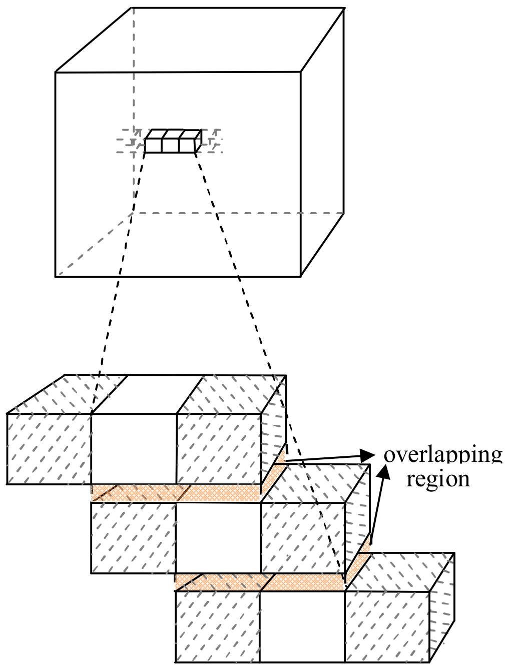

Figure 1: Cache blocking through miniaturizing of domain decomposition results in redundant computations and copies in the overlapping region between the augmented briquettes .

Before pipelining, all the grid planes of an augmented briquette are in the same stage at any given time. After pipelining, the grid planes are in different stages. Each stage is computing on exactly one plane. It is similar to an assembly line where a product assembly happens in several stations. However, there is only one worker manning all the stations. By interleaving the update of several grid planes from different briquettes, the partial results of the computations can be reused. It improves the temporal locality between the stages, and leads very naturally to the memory reduction optimization called maximal array contraction explained below.

The primary motivation for pipelining is *reuse*, not *throughput*. Since the pipelining is very streamlined, the data access pattern in the main memory is deterministic and we can attempt to prefetch the briquettes. Prefetching is done as an assignment operation between the sub-domain in the main memory and temporaries residing on-chip. In the code expression, a new briquette is fetched while the present briquette is being unpacked and progressively updated. At the same time, an old briquette is finished up and sent back to the main memory. Although one briquette is unpacked at a time, many previous briquettes may be living in the pipeline depending on the pipeline length. In general, at least 3 briquettes are needed in play within our cache workspace at one time.

Like all the other pre-compiler transformations, *pipelining-for-reuse* operates only on the compute region. The compute region is well contained within a loop which updates a line of briquettes we call *grid pencil*. It is identified by the cPPM$ PIPELINE directive placed before the loop over a grid pencil. The compute region may contain procedure calls which we inline. After *subroutine inlining*, the compute region has a collection of loop nests that perform the complete update of an augmented briquette. The loop nests contain an inner SIMD vector loop over a grid plane and an outer loop over the planes in the augmented briquette. CFD codes can have this general structure because of causality. The outer loop is pipelined over the planes in the augmented briquette by merging them into a special loop with only *m* iterations where *m* is the length of a briquette. The outer loops are identified by the cPPM$ LONGITUDINAL LOOP directive. To merge the loops, we need to know the number of ghost cells on each side. Currently, annotations are required to specify the number of ghost cells. However, they are not needed as the tool evolves.

2.1.4 Memory reduction through maximal array contraction

The maximal array contraction takes advantage of the property that the partial results of computations have short lives. Most of them do not live beyond a few pipeline stages. The partial results that are alive steadily increase in number during the first half of the pipeline. Similarly, they decrease during the latter half. It means that the circular buffers for the partial results do not have to be as long as the total number of pipeline stages. They can now be made even smaller. The maximal array contraction constructs the smallest possible circular buffer for each partial result. No further memory reduction can be performed on the circular buffers.

We perform array liveness analysis across the whole computation region to determine the life of a partial result. However, we do not perform any liveness analysis outside the region. The computation region is nothing but the pipeline region we operated on in the previous section. Since we inline all the subroutines in the compute region, the whole compute region is visible to the liveness analysis. It would be extremely difficult to do this analysis if pipelining-for-reuse converts the temporary arrays into circular buffers. In our implementation, we instead apply maximal array contraction during pipelining-for-reuse. We only need to look at the subscripts of the temporary arrays corresponding to the pipelined outer loops as identified by the LONGITUDINAL LOOP directive. The subscript adjustments performed in pipelining-for-reuse immensely simplifies the liveness analysis. The inliner retains the call site names of interprocedural variables so that the code is very readable. It also avoids the need of alias analysis for the liveness analysis.

The array contraction techniques usually reduce the dimensions of an array. They do not exploit the opportunities to reduce the size of array dimensions as opposed to only eliminating the array dimensions. The array contraction focuses only on the performance critical region of the application where the computation happens. With such an implementation, all the temporaries needed to update the briquettes can be reduced.

2.2 The structure of CFD Builder

The CFD Builder comprises of three main components, as shown in Figure 7: the pattern library, the intensifying precompiler, and the framework generator. The pattern library comprises of update patterns that specify how to update the data layout. The precompiler optimizes the pattern implementations. The framework generator supplies the code infrastructures for the different patterns. It is still under development. The precompiler in itself comprises of two parts: the frontend and the backend. The frontend performs code transformation to eliminate redundant computation and redundant copying by pipelining, and it performs memory reduction through the maximal array contraction. The backend generates platform specific SIMD and DMA instructions.

2.2.1 Pattern library

A grid update pattern specifies the order in which the briquettes are updated in a sub-domain. Here, we focus only on one pattern called the *grid pencil update* that forms the foundation for the other patterns and the transformations.

Grid pencil update: It is a fundamental pattern of grid update, which is widely applicable in CFD and possibly beyond. In domain decomposition, a single MPI process updates a sub-domain. It is also true with grid pencil update. All the briquettes in a sub-domain are updated by the same process. There is no need for any message passing within the node to update the individual briquettes. There are usually multiple OpenMP threads per MPI process updating the sub-domain. *Grid pencil update* is built for algorithms with directionally split sweeps. In such algorithms, the subdomain is updated in one direction at a time along the x, y, and z directions. The subdomain can be decomposed into pencils of briquettes along the sweep direction. Each pencil is a line of briquettes from one end of the sweep to the other. The OpenMP threads update one pencil at a time each. At the end of a sweep, the threads update the pencils in the new sweep direction. The briquettes are transposed for the next sweep direction to avoid strided accesses and to perform SIMD vectorization. The transposes are efficient because they are performed while the briquette is still on the chip. Processing one pencil at a time is very natural to perform pipelining-for-reuse. A subtle but important point is that SIMD vectorization is built into the grid pencil update and pipelining. The briquettes are processed one grid plane at a time. Each grid plane is a vector operand. The grid plane can be computed with the SIMD instructions.

2.2.2 Intensifying precompiler

The objective of the precompiler is to optimize the pattern implementations. It currently supports only *grid pencil update.* It increases the computational intensity by eliminating the redundant copies and reducing the workspace through the maximal array contraction. It consists of a front end and a back end. The front end is geared towards improving the computational intensity. The back end provides performance portability across the different architectures by generating the appropriate SIMD and DMA instructions where applicable. The precompiler heavily relies on annotations and directives to perform the code transformations. We built both the front end and the back end using ANTLR [2]. We have plans to migrate to ROSE [3].

- *Front end:* The front end performs four main operations: inlining, pipelining-for-resuse, maximal array contraction, and prefetcing. The pipelining interleaves the computation with different stages of a briquette update. Most often, the computations are spread across multiple procedures in the input code. Inlining is essential to bring them together before pipelining. The liveness analysis for the maximal array contraction is simplified by retaining the variables names during inlining and making assumptions based on the update patterns. Although the output from the translator is ugly, we try to make it readable by retaining the variable names during inlining and not creating new temporary variables for the circular buffers during the maximal array contraction. Currently, the front-end only supports Fortran 77. It should be noted that the intensifying precompiler only transforms the computation region that is relatively smaller than the rest of the code. Re-writing just the computation region in Fortran 77 may be unpleasant, but not impossible. ROSE has robust parsers for Fortran, C, and C++.

- *Back end:* Since utilizing the SIMD engines is key to performance, a back end is built to generate the SIMD instructions for the different architectures. It can generate SSE, AVX, and SPU intrinsics for Cell. It provides performance portability across different architectures and the native compilers [4]. The back end also generates the DMA instructions for the Cell architecture. It relies on annotations and directives to guide the code generation.

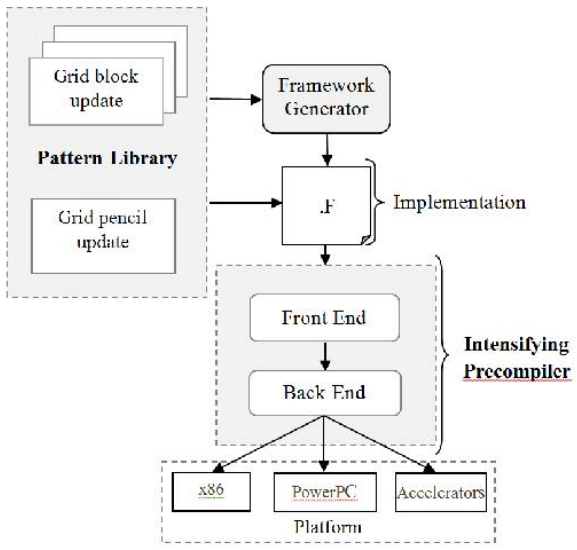

Figure 7: CFD Builder

2.2.3 Framework generator

The hierarchical data layout makes programming hard. In our experience, grid block update is found to be even more difficult to implement. We have come to the conclusion that it is worthwhile to offer our pattern implementations along with the MPI layer through the CFD Builder. The library writers can then experiment with the different patterns and parameters for their algorithms with no effort. The concept is very similar to generative design patterns [5]. The framework generator to support it is still under development.

2.3 Evaluation

The CFD Builder addresses codes based upon numerical algorithms that use grids which cover a 3D domain in physical space. CFD codes use grids that are logically uniform or unstructured. Many unstructured grids have very regular local structure. We believe that these codes can still use the techniques that our code translation tools address. This can be accomplished by taking each cell of such a grid and chopping it up into, say, 4^3 cells to produce a tiny region of grid logical uniformity. This strategy can also be used to handle adaptively refined meshes (AMR). In this study, however, we only focus on uniform Cartesian grids in order to show the benefits of this approach in a simpler context. We have used three codes for the demonstration. Using PPM gas dynamics codes, a code module called PPM advection [6] is extracted for evaluation. This algorithm involves an enormous amount of reuse of intermediate results of the computation, so that many flops are performed for each word of data that is read from or written to the off-chip memory. This algorithm gives a good illustration of the proposed code transformations while involving relatively little code. From the CM1 weather code [7], a code module called Runge-Kutte advection [8] is also extracted for evaluation.

Both code modules has been built into full applications by placing them within the same code framework as full CFD code for testing. The computation is thus carried out in a sequence of six 1-D passes in a repeated pattern of xyzzyx. It should be noted that the messages are passed in all the three dimensions for every single pass. The above two algorithms do not need messages for more than one direction in a pass.

The multi-fluid PPM code is based on the original Piecewise-Parabolic Method [9] and PPB advection scheme [10]. This code is set up to solve an inertial confinement fusion (ICF) test problem described in [11]. This is a fully functional simulation code, which has scaled on NCSA's new Blue

Waters machine to over 730,000 threads running on over 24,000 nodes. The computationally intensive portion of this code that is transformed by CFD Builder consists, after all comments are removed, of 6,082 Fortran lines. This consists of a single subroutine that makes calls to 26 other subroutines. After CFD Builder transforms this block of code, it consists of just one subroutine with 6121 lines of Fortran (with no comments).

2.4 Results from CFD Builder

Experiments are performed to compare the performance of cache blocking through miniaturization of domain decomposition expression and the CFD Builder output for the two algorithms. The cache blocked code expression is the input to CFD Builder. The input expression is called Fortran-W (FW) and the output expression as Fortran Intensified (FI).

Experiments are run on a dual quad-core Intel Xeon x5570 CPU node with a peak performance of 23.44 Gflop/s per-core in 32-bit precision. Intel Fortran v9.1 (ifort) and GCC 4.7.1 (gcc) compilers are used. The options –O3 and –xT –tpp7 are used for ifort and –O3 option for gcc. In addition other optimization flags are set for ifort relating to 32-bit precision. The Intel v9 compiler is used in this study because the Intel v12 doesn’t vectorize all the loops.

SMT is also used by oversubscribing two threads to a processor core. Each Intel Nehalem5570 has four physical cores with hyperthreading enabled. Both sockets in a node are stressed by spawning 8 and 16 OpenMP threads. The computational workload is equally shared among all the threads.

The compute regions of the three codes are measured using CPU_Time. We see performance improvement in general with FI. One of the reasons is that FI does fewer flops than FW. For example, PPM FW performs approximately 454 flops per cell at every time step. However, PPM FI performs only 269 flops per cell. Similarly, the multi-fluid code performs about 5196 flops in FW and 3220 flops in FI. Although only 50-60% more flops are done, performance improvements are obtained of up to 3.77 times with FI. This is mostly due to the benefits of the memory size reduction optimizations. The benefit is bigger for a more complex code. The speedups are better for the multi-fluid codes than the simple advection algorithms. SMT does seem to help also. It is generally run about *15%* faster with SMT turned on. GCC fairs poorly because it does not vectorize all loops.

The percentage of peak performance for the FI codes of the two algorithms is also measured. Since FW performs more flops for the same amount of work, it is not a good candidate to measure the percentage of peak performance. The FI code expression does not have any redundant computations. The flops are measured in Cray-1 style. Each divide is counted as 4 flops. The adds and multiplies are counted as 1 flop. It is found that both the algorithms are capable of executing above 20% of the processor peak. PPM is a compute intensive algorithm and is a natural candidate for the pipelining strategy. However, it is encouraging to see that a computational less intensive algorithm like RK advection can run so fast.

2.5 Summary

The combination of the data layout, pipelining-for-reuse and maximal array contraction, and carefully designed grid update patterns increases the computational intensity of CFD algorithms. However, the code expression necessitated by this combination of techniques is highly unreadable and unmaintainable. We have built a source-to-source precompiler to perform the tedious code transformations for the implementations of our fundamental grid update pattern. It has shown its utility of achieving a high percentage of processor peak on two different numerical schemes for advection codes. It showed that CFD Builder is a good productivity tool for building efficient libraries in CFD.

This project has been partially supported by other fundings from NSF and DOE. Two PhD theses [12] [13] have been completed from this project.

3. A Domain-Specific Compiler for Cryptography

Much of the state-of-the-art in public-key cryptography is based on arithmetic on elliptic and hyperelliptic curves over finite fields. In a typical primitive, almost all of the computing time is spent in group operations on the curve, which in turn comprise a few operations each in the underlying finite field.

All current speed records for elliptic-curve cryptography are set by hand optimized assembly implementations, in particular of the finite-field arithmetic. Libraries for big-integer arithmetic usually fall short in performance because they do not make use of the special form of prime moduli typically used in elliptic-curve cryptography. The library-generator `mpFq` is a huge step forward but still optimizes separate arithmetic operations (such as multiplication or addition) instead of sequences of operations. It also does not handle multiple independent streams of operations and can thus not make best use of vector instructions or many-core platforms, and is limited to the x86 and AMD64 architectures.

The target of this project is to develop a way to generate high-speed software for finite-field arithmetic on various platforms including various types of parallelism on hetergeneous multi-cores. Since this proposal specifically targets crypto, most code once generated will be running for hundreds of CPU-years (a realistic assumption for cryptographic software library implementations and cryptanalytic efforts), and it assumes ample computing resources are available before we determine which is the best code to use.

3.1 Semi-automatic code generation for high-speed crypto

The first sub-project consists of a compiler that is able to generate functioning fast code code for a given target function on a given target architecture. The initial test case is the Ed25519 Edwards curve arithmetic on AMD64 processors; we are familiar with hand-optimized assembly code for this example and thus understand the desired compiler behaviour very well. This sub-projects requires the following subtasks:

Specification of the input language. The input language needs to be able to specify a finite field, arbitrary sequences of operations in that finite field, and the number of independent data streams this sequence is operating on. The language does not have to support any loops or control structures.

Determine representations of elements. Typical field sizes for ECC have between 160 and 512 bits, standard computer architectures do not support arithmetic on integers of that size directly. It is important to determine a good representation of field elements. The obvious choice is to express elements of a 256-bit field can be as 4 64-bit integers, but often using a smaller radix than 2^{64} can avoid frequent carries and thus result in faster code.

This step should already take into account the structure of the field (in most cases, the bitstream structure specific to the modulus) because reduction algorithms using this structure depend on the representation.

Enumeration of Optimization Choices. To get fast code we require optimization on various levels. Note that we consider compile time as a minor issue as per description of the problem above. On each level of the compiler it is possible to make different choices: different *representation of finite field elements in machine words*, different *multiplication algorithms* (in particular schoolbook and Karatsuba multiplication strategies), different *modular reduction strategies* (either based on the structure of the modulus or general Montgomery reduction, complete or lazy or partial).

This is clearly an exponential problem so early-abort strategies are needed to not generate combinations that either will not work or known to be slow. Another optimization required for high-performance code is instruction scheduling, the optimal solution of which together with register allocation (including spill-code generation) is NP-complete. Two techniques from the literature are tried for our mostly branch-free long sequences of instructions: linear programming and constraint-programming techniques.

Actual Code Generation (Register Allocation and Spill). Translating high-level finite-field operations into machine-level instructions (in any of the representation determined in the previous step) is needed The `qhasm` programming language is used as intermediate code, which has unified syntax support for various architectures and makes cross-platform development easier. `Qhasm` also handles allocation of registers to an arbitrary number of register variables *if an allocation without spills exists*. Most intermediate code will fail translation to assembly through `qhasm` because it uses too many live registers. We aim to start with spilling variables to the stack in a simple greedy approach and bring in better heuristics as we go.

3.2 Ensuring correctness of code via formal methods

The second subproject is an attempt to ensure that the optimization choices and the final generated code for a given primitive is "correct". This may differ from the usual idea of correctness, in that many crypto applications require side-channel resistance. This means that no conditional branches or secret indices can be used. We use both the main branches of Formal Verification in our attempt.

3.2.1 Theorem proving

One programming language based approach to correctness would be to ensure that the compiler is correct. We will attempt to give an operational semantics of the input language, and an operational semantics of the subset of `qhasm` we compile the language to, and prove that the compilation preserves the semantics. A formal proof of the compiler can be carried out by proof assistants including `Coq` or `Agda`.

Proof of the compiler correctness shall be developed together with the compiler itself. Large-scale projects of this nature has been done before — a well known project in INRIA has built a verified C compiler using `Coq`. To begin with, we will build a small prototype compiler to check whether this approach is flexible.

Guaranteeing time and resource usage is a challenging task. There has been experimental type systems that guarantee time and space usage — that is, for a program to type check have to, for example, have a constant or polynomial complexity, or use constant space. It remains to see whether one may enforce such a type system either on the source language or on an annotated version of the `qhasm` assembler.

3.2.2 Model checking

Model checking is a fully automatic approach that tests if a finite state system or a finite abstraction of an infinite state system obeys some given properties defined in the form of logics or automata. In case that the system does not satisfy the desired property, a diagnostic trace that explains why the property is not satisfied by the system will be returned. Such trace is very useful for system designers to find the source of the buggy behaviour.

Bit-level precision is particularly important in the analysis of the compiler. Such analysis is usually simple but tedious. It might involve a large number of case splits for different corner cases, although most of the subcases are trivial. Model checking is best suit for such an application. We plan to use model checking to prove properties of individual bit-manipulating functions of the compiler. Theorem proving can then work on top of the proven properties and thus avoid the tedious part.

3.3 Major results

3.3.1 Prototype compiler and code generation

1. On an Intel Nehalem, our code generator is 1.07× the speed of the hand-crafted assembly code from [16]. These routines have the same functionality (they are time-independent and contains no data-dependent table lookups).

2. We can generate Westmere, Sandy Bridge and Ivy Bridge code for Intel processors, which is within 5% of the speed of the hand-crafted assembly code, and code with ARMv11 and ARM Neon instructions within 10% of the speed of the hand-crafted assembly code can be produced.

3. Our code generator were able to generate code which is >70% the speed of $\mathtt{mpF_q}$ code, using the same representation – except that the $\mathtt{mpF_q}$ code does not have time independence.

Our code was contributed and is included in the current codebase of the Ed25519 signature scheme, and is part of the code described by [15], the complete version of [16]. *It is usually considered for handcrafted assembly to have a 1.5× advantage at least over generated code***.** Coming this close to handcrafted assembly is a rare feat.

3.3.2 Formal verification

We have worked on the formal verification of high speed cryptographic code by submitting the results of code generation to formal methods, particularly *model checking*. The procedure is to translate `qhasm` assembly code to a representation that is suitable for `Boolector`.

1. We are able to verify the `maddmod` (multiply-with-add, modulo) operation as used in Ed25519 ($A \leftarrow A + BC 2^{255} - 19$) in the radix- 2^{51} redundant representation, and will complete the verification of the same in the radix-$2^{25.5}$ redundant representation used in ARM and Sandy Bridge/Ivy Bridge.

2. We discovered a bug in Ed25519, caused by a very rare carry bit being sent into the wrong limb. **Such an error is extremely rare in a work involving the meticulous Daniel J. Bernstein and his students.** The reason that this error went unnoticed despite paranoid regression testing was because the probability for error was way smaller what could have been caught by random tests, but can always be found by SAT solvers given sufficient time.

   We also discovered that, in a similar vein and using similar model-checking methods, one carry in the Ed25519 codebase is unnecesary when the input is in canonical form, but not if it is in a semi-reduced form, which allows us to save time if program space is not a problem. Again, this is unheard of in the work of Bernstein et al, and likely only possible with automated checkers.

3. We have not yet verified the radix-2^{64} non-redundant representation and have identified the key reason for this. The main reason is the use of redundant representations allow us to delay carry chains. The presence of long carry chains force interdependence of the bits under examination; this raises the number of state bits and pushes the problem outside the capabilities of `Boolector`.

The fact that we can effectively verify formally long assembly language routines in long integers is in itself already a breakthrough. In [14], the authors even stated that “PKC (public-key cryptography) is particularly hard because it involves multiplication and modular exponentiation on long bit-vectors. Hence, the bit-level representation of any PKC algorithm is usually so huge that such equivalence problems are too hard for current SAT solvers, ...” Indeed, we were unsure that this is even doable as we began our work. To the best of our knowledge, [14][15] are the only published works that attempt to verify cryptography algorithms. Such results however are still in early stages; only short papers are published.

3.3.3 Formal verification approach

A formal verification of the Ed25519 code for any single platform would already be a breakthrough in formal methods; there has been (to the best of our knowledge) no publication on a full verification of a productivity-level assembly language routine on the order of 100,000 instructions to date.

We hope to complete the following:

- Semi-automated model checking for big-integer modular arithmetic assembly routines.

  At the moment, we are able to complete the model checking only through the process of *“tagging”*: At various checkpoints, we add to the assembly comments that

  - serve as “assertion” statements so that if each assertion is valid then the result is correct, and

  - also serve to tell, in somewhat interleaved code (common to all processors with long latency) is split into more or less independent threads of execution.

What we do is

  - to verify that each thread, from a proper starting state, will arrive the correct concluding state, and
  - verify that the threads do not intefere with each other (i.e., two sequences have the same result).

  At the moment all the tagging is done by hand. We will need support from the code generator (compiler) to achieve semi-automation.

- A verification of a practical-sized full-carry full-reduction big-integer (about 256 bits) modular arithmetic assembly routine.

  The natural "state" involved in such a thread of execution is larger than what we can expect to be verifiable with model checking. To verify such a procedure, we need to subdivide the threads further into smaller chunks of code and then show that this subdivision achieves the goal we want. This may require us to introduce "theorem proving" methods into the mix.

- At the moment, we only do prime field code. In theory, binary code should be much easier to achieve; the procedure reduction in "tower" fields should be no different, in principle, than for reduction and carry in prime fields. The challenge is that such field arithmetic is often batched or vectorized. Again proper tagging, for vectorized arithmetic, is required.

3.3.4 Code generation and compiler techniques

Better Language for Code: right now, we can choose different representations, but instructions of the critical inner loops are still crafted, or "placed" by hand because we cannot (so far) automate the process when it comes to producing fast code.

Further, much faster code are written in a batched way, i.e., we can only do something fast if done *X* at a time. This is done in an extremely rudimentary way right now, i.e. our code generator can only accept a very minimal python-like syntax (compatible with the *Explicit Formulas Database* [18]).
At the moment, we are experimenting in the code generation for our Hydra co-processor, we have taken to use Haskell for code generation, then providing for translation with Haskell as a Functional Language. This allows more natural extensions to the language to be built, and allows natural self-assembly of small functional units, which now we can do (only) for 2-way and 3-way Karatsuba.

LLVM, a Better Backend for Code: We cannot stress enough the need for automated generation of code snippet and assembly. These is impossible with the current approach or with current `gcc` backend. We are planning to make the difficult switch over to LLVM for this task.

We are fortunate that Mr. Shang-Yi Yang, who will be with us for 3 years, is able to start his early training in LLVM directly from Marvell in this aspect. There is no better time or opportunities for this venture since coders with such know-how will be in extremely high demand, and only because our lab did some security checking for them to receive such favors from them.

3.3.5 Main choke-points of work

Cryptographic Engineering

- As we discovered halfway into this project, there is no really existing good language for this work. We concede that our original specifications (which is essentially a primitive python) is way too primitive, and we essentially will have to come up with a domain-specific language. Thankfully, we do have a real expert Dr. Mu in the field around to coach us. At the moment we lean toward using something similar to or based on Haskell, but

- The backend of the code generation needs to be reworked. We are now working with LLVM and hoping to morph it into something that provides multiple possible results of the compilation (to time and compare). The intersection of cryptographic programming with lots of finite field code generation with expert compiler backends are for the moment an empty set around the world.

Formal Methods

- Currently, our methodology is not fully automatic. Particularly, we need to insert properties and decompose verification tasks manually. Subsequently, our semi-automatic technique cannot be applied to numerous cryptographic programs generated by cryptographic engineering. We plan to enrich our domain-specific language with property specifications. High-level properties about programs written in the domain-specific language can therefore be propagated to low-level programs. Verification tasks can be automated by guidance from high-level properties.
- Our modular methodology can alleviate computational complexity in SAT solvers sometimes. There are known cases where our methodology fails to verify. Improving the scalability of bit-vector theory solvers will be necessary. We are in favor of combining other techniques with bit-vector theory. Particularly, term rewriting and theorem proving may further simplify verification tasks. In collaboration with INRIA researchers, we are hopeful to develop such a hybrid methodology for verifying cryptographic programs.

3.4 Major results

Cryptographic Engineering

- First year: a language specification and a compilation result (possibly including partial working by hand) that actually defeats hand-craft code in the best of class, such as those produced and constantly updated by Bernstein et al in [16] and Longa et al in [19].
- Second year: a compiler tha succeeds in outputting, among a reasonable number of alternatives, a best-of-class work considered above.

Formal Methods

- First year: an assertion language specification and its translation to low-level properties that enable automatic verification of programs generated by cryptographic engineering. The goal is to fully automate our modular methodology for the verification of cryptographic programs.
- Second year: a hybrid methodology that decomposes verification tasks and dispatches them to techniques such as bit-vector theory solvers, term rewriting, and theorem proving. We aim to further extend our methodology to cryptographic programs that are hard to verify.

3.5 Impact of Work

Cryptographic Engineering

The importance of side-channel resistance and high speed in Cryptographic Engineering is recently being recognized again. The Ed25519 code, of which we played a small part, is currently gaining some traction and we are trying our best to be part of its future by contributing to its speed and security.

Such a contribution will create a few good papers (probably at CHES or JCEN), doubtlessly increase the acceptance of use of high-speed crypto code, and more importantly in getting more mileage out of our work in multivariate and lattice-based cryptography, since the eventual product will apply across many fields and affect almost all public-key cryptography.

Formal Methods

Formal method community is always looking for new applications. Verifying cryptographic programs is certainly an unexplored domain with significant impact. Our work will also give new insights and guidance to the verification research community. Different from general beliefs, our modular methodology developed for verifying multiplication programs over Ed25519 shows that SAT solvers are capable of checking arithmetic operations with hundreds of bits. Our new methodology will help the research community develop new heuristics for solving arithmetic problems.

3.6 Potential competitors and collaborators

Cryptographic Engineering

There are several groups that might be in direct competition with us in the field of high-speed cryptography:

- Microsoft Research in the section headed by Kristin Lauter, this section contains three capable crypto-programmers who excel for finite fields: Dr. Joppe Bos, Dr. Patrick Longa, and Dr. Michael Naehrig; due to both the philosophy of Microsoft (not opening source code and patents) and their history with us (enmity from Longa and to some extent Bos) this group is both capable and competitive party. However, none of them are compiler-oriented people, each of them prefer to deal with bare metal directly. They also do not focus on side-channel resistance.

- The Netherlands group of Prof. Daniel J. Bernstein, Prof. Tanja Lange, and Prof. Peter Schwabe, whose group are all long-time friends and collaborators. Again they are also assembly- and bare-metal oriented people. However, Prof. Bernstein has promised us an update of the `qhasm` tools and constant updates of their architectural improvements, and that will be very helpful to us.

- The French INRIA group headed by Pierrick Gaudry, whose group already produces the $\mathtt{mpF}_{\mathtt{q}}$ library. They recently declined a proposal to work together, but they are at most neutral and is headed away from this direction.

Thus, we think that we still have a possible lead against our most competitive adversaries.

Formal Methods

- Professor Jorge Sousa Pinto at Universidade do Minho, Braga, Portugal and HASLab / INESC TEC applies rewriting techniques to simplify problems before verification. The technique however is not very mature and does not verify low-level cryptographic programs.

- Several research groups are working on solvers for bit-vector theory. For instance, Armin Biere (Johannes Kepler University, Austria), Nikolaj Bjorner (Microsoft Research), Alessandro Cimatti (Bruno Kessler Foundation, Italy), and Chung-Yang (Ric) Huang (National Taiwan University). Our test cases generated from cryptographic programs are challenging for all state-of-the-art solvers. Collaboration will help solver developers to develop new techniques for solving large bit-vector problems.

- We have close connection with several INRIA researchers specialized in the proof assistant `Coq`. Their expertise will be useful to compensate limitations of bit-vector solvers.

3.7 Involved Personnel

Dr. Peter Schwabe graduated from Technical University of Eindhoven and is an expert in high-speed cryptography programming. He worked with us through most of 2012 as a postdoc and then left to take up an assistant professorship at the prestigious Raboud University

Nijmegen in the Netherlands. Dr. Schwabe continue to be a collaborator even though somewhat occupied with his new teaching duties.

Dr. Hsin-Hong Lin graduated from National Taiwan University and is an expert on model checking. He worked with us through most of 2012 and until April 2013, when he left for Fukuoka, Japan as a visiting scholar on a JSPS scholarship.

Dr. Ming-Hsien Tsai took over from Dr. Chang in mid-2013 before graduated from National Taiwan University this year. He is a young up-and-coming expert on formal verification and languages, and his impressive resumè include joint work with Turing Award laureate E. Clarke at CMU).

Mr. Shang-Yi Yang was a hard-working MS grad student who decided on National Service and continuation onto a Ph.D. program in our lab. He has worked on assembly programming projects starting in September 2012.

Mr. Robert Fitzpatrick is a Ph.D. student at Royal Holloway (City College of London) who has started to work with the lab and has committed to work at IIS as a postdoc starting in early 2014 after he receives his Ph.D.

4. References

[1] D. J. Bernstein, *Qhasm: tools to help write high-speed software,* http://cr.yp.to/qhasm.html

[2] T. Parr, *The definitive ANTLR reference,* Pragmatic Bookshelf, 2007*,* http://www.antlr.org/

[3] D. Quinlan*, ROSE: Compiler Support for Object-Oriented Frameworks,* Proceedings of Conference on Parallel Compilers (CPC2000), Aussois, France, January 2000.

[4] P.H. Lin, J. Jayaraj, P. R. Woodward, P. Yew, *A Code Transformation Framework for Scientific Applications on Structured Grids*, Technical Report 11-021, UMN Computer Science and Engineering Technical Report, Sep. 2011.

[5] K. Tan, D. Szafron, J. Schaeffer, J. Anvik, and S. MacDonald, *Using Generative Design Patterns to Generate Parallel Code for a Distributed Memory Environment*, Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2003), San Diego, CA, June 2003, pp. 203-215.

[6] P.R. Woodward, *A Complete Description of the PPM Compressible Gas Dynamics Scheme*, LCSE internal report available at *http://www.lcse.umn.edu/ppm/PPM-for-ILES-2-4-05.pdf*

[7] http://www.mmm.ucar.edu/people/bryan/cm1/

[8] L. J. Wicker and W. C. Skamarock, *A time splitting scheme for the elastic equations incorporating second-order Runge-Kutta time differencing*. Mon. Wea. Rev., 126, 1992–1999

[9] P. Colella and P. R. Woodward, *The Piecewise-Parabolic Method (PPM) for Gas Dynamical Simulations*, J. Comput. Phys. 54, 174-201 (1984).

[10] P.R. Woodward, *Numerical Methods for Astrophysicists*, in Astrophysical Radiation Hydrodynamics, eds. K.-H.Winkler and M. L. Norman, Reidel, 1986, pp. 245-326, also available at *http://www.lcse.umn.edu/PPMlogo*

[11] P.R. Woodward, J. Jayaraj, P.-H. Lin, M. Knox, C. L. Fryer, G. Dimonte, C. Joggerst, G. M. Rockefeller, W. Dai, R. J. Kares, and V. A Thomas, *Simulating Turbulent Mixing from Richtmyer-Meshkov and Rayleigh-Taylor Instabilities in Converging Geometries using Moving Cartesian Grids,* LA-UR-12-23994, also available at *http://www.lcse.umn.edu/NECDC2012*.

[12] P.H. Lin, *Performance Portability Strategies for Computational Fluid Dynamics (CFD) Applications on HPC Systems*, PhD thesis, Department of Computer Science and Engineering, University of Minnesota at Twin-Cities, June. 2013

[13] J.Jayaraj, *A Strategy for High Performance in Computational Fluid Dynamics,* PhD thesis, Department of Computer Science and Engineering, University of Minnesota at Twin-Cities, August 2013

[14] I. Abal, A. Cunha, J. Hurd, and J. S. Pinto, *Using Term Rewriting to Solve Bit-Vector Arithmetic Problems*, SAT 2012 (The International Conferences on Theory and Applications of Satisfiability Testing), Lecture Notes in Computer Science, Volume 7317, pp 493–495. A. Cimatti and R. Sebastiani, Eds.
[15] I. Abal and J. S. Pinto, *Towards a Mostly-Automated Prover for Bit-Vector Arithmetic*, Sixth International C* Conference on Computer Science and Software Engineering, pp 132–133. A. Almeida and S. Mudur, Eds. 2013. ACM.
[16] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, *High-speed high-security signatures*, Journal of Cryptographic Engineering **2:2**(2012), pp. 77–89. See also ePrint Archive: *http://eprint.iacr.org/2011/368.*
[17] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, *High-speed high-security signatures*, CHES 2011 (13th Workshop on Cryptographic Hardware and Embedded Systems, September 28 – October 1, Nara, Japan), Lecture Notes in Computer Science, Volume 6917, pp. 124–142.
[18] D. J. Bernstein, T. Lange, *Explicit Formulas Database for Elliptic Curves*, *http://www.hyperelliptic.org/EFD*
[19] A. Faz-Hernandez, P. Longa and A. H. Sanchez, *Efficient and Secure Algorithms for GLV-Based Scalar Multiplication and their Implementation on GLV-GLS Curves*, Cryptology ePrint Archive: Report 2013/158 *http://eprint.iacr.org/2013/158.*